Crafting An Ada Executive

for the

Intel 80960 Extended Architecture

14 May 1992

Chak Sriprasad
Intel Corporation
5000 W. Chandler Blvd.
Mail Stop SP1-82
Chandler AZ-85226
csriprasad@AZ.INTEL.COM
(602)554-4030

1. Introduction

The Joint Integrated Avionics Working Group (JIAWG) selected the Intel 80960 Extended Architecture as a standard Instruction Set Architecture for 32-bit processors. The key to the promulgation of this JIAWG standard is the availability of quality Ada compilation systems. A generic Ada-960MX Cross-Development System consists of a set of software tools that compile, link, and debug Ada programs that are targeted to the 80960 extended architecture. Part of Ada-960MX is a **runtime system** (RTS), a set of predefined routines for any program generated by the Ada-960MX compiler. Part of the RTS that directly interfaces to the architecture is the Ada Executive (exec) which performs the kernel or executive functions required to implement Ada semantics. It supports Ada tasking, delays, interrupt management, dynamic storage allocation, and exception raising. This paper is a discussion of how these features may be implemented on the 80960 extended architecture.

The full functionality of the Ada Executive (exec) is based on the Ada Runtime Environment Working Groups (ARTEWG) (a subgroup of the Association for Computing Machinery, Inc., Special Interest Group for Ada) published "A Model Runtime System Interface for Ada" (MRTSI)[1]. The MRTSI describes a model interface between the code generated by an Ada compiler and the executive portion of the runtime system. This interface is used as the basis for specifying the Ada Executive.

The exec also provides some real-time extensions to the Ada runtime. This is based on a set of features selected from the "Catalog of Interface Features and Options" (CIFO)[2], which is another ARTEWG proposal. The selected CIFO features consist of: basic entries that support task identification and querying of task attributes; task scheduling control including dynamic priorities, entry and select criteria, task suspension and holding and time slicing; Interrupt management functions that allow selective enabling and masking of interrupts.

The 80960 extended architecture features very high levels of hardware-enforced data security, and support for object-oriented programming in hardware. It also offers virtual memory management and multitasking support. Specific instructions and data structures are dedicated to provide a complete multiple process management capability, including priority-driven process scheduling, process timing and interprocess communication. The exec strives to best exploit these architecture features.

The following section gives a brief overview of the 80960 Extended Architecture. A full discussion of the architecture is beyond the scope of this paper. This paper assumes a fair knowledge of this architecture.

1.1. Overview of the 80960 Extended Architecture

The 80960 extended architecture is an extension to the core, numerics and protected series of architectures. The core architecture directly addresses a 32 bit physical address space and includes a large register set. It supports signed and unsigned integer arithmetic, many bit oriented operations, support for procedure calls, interrupt handling, fault handling and debugging. The numerics architecture adds floating-point data types, registers and instructions to the core architecture. It supports 32-bit single precision, 64-bit double precision, and 80-bit extended precision floating point types. The protected architecture adds virtual memory management support including paging and segmentation. It provides on-chip support for Ada multi-tasking including automatic process scheduling and dispatching. Each process (task) has its own protected address space.

The extended architecture includes all protected architecture features and adds object based memory organization and protection[3][4]. Different software subsystems can have different address spaces or domains, providing protection between parts of a program as well as between processes. Other added protection features include object type checking, checking of access rights in object pointers or access descriptors (ADs) and protection of ADs against unauthorized changes. Objects are typed and protected memory segments. They can only be accessed with ADs. The extended architecture supports tagging. Tagging associates a 33rd bit, the tag bit, with each 32-bit memory word. This tag bit provides the distinction between data words and words that hold ADs. In addition, the tag bit prohibits the possibility of forged pointers to protected areas within memory, thus providing a high level of data security within the architecture.

The physical address space is the 2^{32} byte address space that directly maps to physical memory. The virtual address space contains all the bytes in all the objects in the system. A 64-bit virtual address has two parts, a 32-bit offset into an object and an AD that references the object.

2. Specification and Scope of the Ada Executive

The following is the set of MRTSI packages and features that embody the core functionality of the exec. Refer to the MRTSI document for a formal definition of the interface.

- package Compiler_Exceptions
- package Machine_Specifics
- package RTS_Abortion
- package RTS_Clock
- package RTS_Delays
- package RTS_Interrupts
- package RTS_Priorities
- package RTS_Queued_Interrupts
- package RTS_Rendezvous
- package RTS_Storage_Management
- package RTS_Task_Ids
- package RTS_Task_Stages

While MRTSI is essentially a compiler independent specification, it does identify a host of cross dependencies among the compiler, the runtime system, and the processor. This paper addresses this aspect alone in section 3 "Interface Characteristics".

The following CIFO packages represent the subset of CIFO features chosen. Once again the reader is referred to the CIFO document for the actual specification.

- package Asynchronous_Task_Holding
- package Complex_Discipline
- package Dynamic_Priorities
- package Interrupt_Management
- package Queuing_Discipline
- package Resources
- package Task_Ids
- package Task_Suspension
- package Time_Slicing
- package Two_Stage_Task_Suspension
- Entry Criteria
- Select Criteria
- Time Critical Section

The specification listed above provides the framework and defines the scope of the exec. The focus of this paper is to identify and define the characteristics of the interface and show how the implementation of the features represented by this specification maps to the 80960 extended architecture.

3. Interface characteristics

The exec interfaces to the compiler and to the 80960 processor. This section identifies the significant aspects of these two interfaces, and the dependencies on the compiler, the Ada executive, and the 80960 processor.

3.1. Compiler Dependencies

The implementation of the exec is mainly in Ada. However, it is likely that some parts of the exec will be implemented in assembler. This requires a full understanding of details that are determined by the compiler, but which affect the exec implementation. These include:

- The data representation of several predefined types like `INTEGER`, `BOOLEAN`, `DURATION`, and `SYSTEM.ADDRESS`, as well as all types defined in the Ada executive.
- Subprogram calling conventions.
- Subprogram parameter passing conventions.
- Function return conventions, for functions returning values of a discrete type.
- The representation of exceptions as data, and the mechanism of raising an exception.
- The implementation of pragma Access_Descriptor to implement Ada access types as architecture access descriptors.
- Interrupt handling code (associated with task interrupt entries) should be located in Region 3. The hardware interrupt mechanism only saves all local register sets, and gives control to the handler under the context of the current executing process. Thus the compiler may need to save all global and floating point registers used in the handler or other subprograms that it calls and restore it prior to returning from the handler.

The compiler vendor should document how

these details are resolved to help integrate the Ada executive with the compiler and runtime systems.

3.2. Ada Executive Dependencies

The dependencies of the compiler on the exec consists of all data types and constant definitions used in specifying the compiler-runtime interface modules. The Ada executive establishes the definition of the following constants and type declarations:

· The representation and constraints of type **STANDARD.DURATION.**

The type DURATION shall have the following characteristics:

Table I - Duration Attributes

Attribute	DURATION value
'DELTA	0.0001 sec
'FIRST	-86400.0 sec
'LAST	86400.0 sec
'SMALL	0.000061 sec

· The following types and constants in package **SYSTEM.**

```
TICK              : constant := 2 ** (-14);
                               -- 0.0000614

-- The processor supports 32 priorities for
-- processes and interrupts. The Ada task
-- priorities are restricted to the range
-- 0..15 in order to allow interrupts to have
-- higher priorities than tasks.
subtype   Priority   is INTEGER range 0..15;

-- The Object table supports 2**26 entries.
-- This type is to index into the object
-- table.
type   Index   is range 0 .. 2**26 - 1;
for    Index'Size use  26;

-- Representation rights to read or write an
-- object's representation
type   Read_Rights   is new Boolean;
type   Write_Rights  is new Boolean;

-- Type rights that are dependent on the
-- object's type
type   Control_Rights   is new Boolean;
type   Modify_Rights    is new Boolean;
type   Use_Rights       is new Boolean;

type   Lifetime       is
   (Global,
    Local);
for Lifetime use
   (Global => 0,
    Local  => 1);

type   Access_Descriptor   is  -- may be
                -- defined in private section
   record
     Object_Read_Rights    : Read_Rights;
     Object_Write_Rights   : Write_Rights;
     Object_Use_Rights     : Use_Rights;
     Object_Modify_Rights  : Modify_Rights;
     Object_Control_Rights : Control_Rights;
     Object_Lifetime       : Lifetime;
     Object_Index          : Index;
   end record;

for Access_Descriptor use
   record
     Object_Read_Rights    at 0 range  0 ..
0;
     Object_Write_Rights   at 0 range  1 ..
1;
     Object_Use_Rights     at 0 range  2 ..
2;
     Object_Modify_Rights  at 0 range  3 ..
3;
     Object_Control_Rights at 0 range  4 ..
4;
     Object_Lifetime       at 0 range  5 ..
5;
     Object_Index          at 0 range  6 ..
31;
   end record;

type   Ordinal   is range 0 .. 2**32 - 1;

type   Address_Mode is (Virtual, Linear,
Physical);

type   Address (Mode : Address_Mode)  is
   record
      case Mode is
      when Virtual =>
         Offset    : Ordinal;
         AD        : Access_Descriptor;
      when Linear =>
         Linear_Address : Ordinal;
      when Physical =>
         Physical_Address : Ordinal;
      end case;
   end record;
for Address use
   record at mod 32;
      Mode             at 0 range  0 .. 7;
      Offset           at 4 range  0 .. 31;
      AD               at 8 range  0 .. 31;
      Linear_Address   at 4 range  0 .. 31;
      Physical_Address at 4 range  0 .. 31;
   end record;

subtype Virtual_address is Address (Mode =>
Virtual);
subtype Linear_address  is Address (Mode =>
Linear);

function Virtual (Lin_Addr : Linear_Address)
   return Virtual_Address;
```

3.3. 80960 processor Dependencies

There are some ways in which both the compiler and the exec depend on conventions about their mutual view of the target machine. They are embodied in the MRTSI package **Machine_Specifics** to facilitate adaptation to specific compilers. The processor specific type definitions are:

- **Init_State**, the initial task state. This is a record of key register values and other task start-up related information.

```
type Init_State is
   record
      Entry_Point : System.Address;
      Storage_Size : Natural;
      Body_Elaborated : Boolean;
      Entry_Criteria :
Queuing_Discipline.Discipline;
      Select_Criteria :
Queuing_Discipline.Discipline;
      Region0 : System.Access_Descriptor;
      Region1 : System.Access_Descriptor;
      Prim_Env_Table_AD :
System.Access_Descriptor;
   end record;
```

- **Pre_Call_State**, the portion of the state of the processor that is not preserved by the subprogram calling conventions.

```
subtype Signed_32  is Integer;
type Pre_Call_State is
   record
      Reg_G0          : Signed_32;
      Reg_G1          : Signed_32;
      Reg_G2          : Signed_32;
      Reg_G3          : Signed_32;
      Reg_G4          : Signed_32;
      Reg_G5          : Signed_32;
      Reg_G6          : Signed_32;
      Reg_G7          : Signed_32;
      Reg_G8          : Signed_32;
      Reg_G9          : Signed_32;
      Reg_G10         : Signed_32;
      Reg_G11         : Signed_32;
      Reg_G12         : Signed_32;
      Reg_G13         : Signed_32;
      Reg_G14         : Signed_32;
      Reg_G15         : Signed_32;
      Trace_Controls  :
Trace_Controls_Words.Trace_Control_Word;
      Reg_SF0         :
Special_Functions_Registers.Special_Function_
Reg0;
      Reg_SF1         :
Special_Functions_Registers.Special_Function_
Reg1;
   end record;
```

- **Task_Storage_Size**, the type used to specify task storage size.

```
type Task_Storage_Size is range 0..2**32;
```

- **Interrupt_Id**, the type used to identify a hardware interrupt.

```
type Interrupt_Id is range 8..255;
```

- **Machine_Exceptions**, the type used to identify a hardware detected exceptions when notifying the exception recovery system.

```
type Machine_Exceptions is new
Fault_Information.Fault_Type;
```

- **Error_Information**, the information associated with a hardware detected error.

```
type Error_Information is new
Fault_Information.Fault_Record;
```

- **Allocation_Descriptor**, the information telling how a dynamic storage collection is allocated.

```
type Allocation_Descriptor is (Linear,
Virtual);
```

4. Implementation Characteristics

Based on the selected set of interfaces the key components of the exec are identified as:

- Task Management
- Interrupt Management
- Time Management
- Storage Management
- Exception Propagation
- Time Critical Section
- Resources

The implementation of these features is discussed in the following sections. The emphasis is more on utilizing processor features and instructions while following a general algorithm that implements the Ada semantics for these features. This has the potential to be expanded to a rigorous design document.

4.1. Task Management

4.1.1. Task Identification

A task is identified by a task control block. This is a record that consists of the architecture defined **process object** and a set of task management fields. An AD created for this extended process object serves as a task identifier. An implementation may restrict access rights on this AD so that user code may not directly access the

task control block. The following Ada code represents the definition of a task identifier:

```
type Task_Control_Block;

type Task_Id is access Task_Control_Block;
pragma Access_Descriptor (Task_Id);
-- The pragma tells the compiler to implement
-- the access type Task_Id as an Access
-- Descriptor.

type Task_Control_Block is
   record
      Process_Info              :
Process_Control_Blocks.Process_Object;
-- Architecture defined data structure

      State                     :
Ada_Task_State;
      Status                    :
Ada_Status_Flags;
      Activator                 : Task_Id;
      Num_Tobe_Activated        : Natural;
      Act_Chain                 :
Activation_Chain;
      Act_Semaphore             :
System.Access_Descriptor;
-- AD to Semaphore object used to
-- synchronize activation

      Task_Type_Descriptor      :
Machine_Specifics.Init_State;
      Parent_Task               : Task_Id;
      Master                    : Master_Id;
      Child_Tasks               : Task_Id;
      Previous_Child            : Task_Id;
      Next_Child                : Task_Id;
      Leaving_Block             :
System.Address; -- Virtual address
      Ref_Dependents_Port       :
System.Access_Descriptor;
-- AD to Port object used to queue completed
-- tasks as reference dependent on this task.

      Delay_Period              : Integer;
      Delay_semaphore           :
System.Access_Descriptor;
-- AD to Semaphore object used to signal
-- a task upon expiration of a delay.

      Entry_Queues              :
Port_Object_Anchor;
-- List of AD's to architecture defined
-- Port Objects for each entry.

      Called_Entry_Port         :
System.Access_Descriptor;
-- AD to the entry queue Port object on
-- which this task is a message waiting
-- to rendezvous (used for timed entry calls)

      Partners                  : Task_Id;
      Next_Partner              : Task_Id;
      Select_Semaphore          :
System.Access_Descriptor;
-- AD to the Semaphore object on which this
-- task waits when none of the open select
-- entries are called.

      Select_List               :
RTS_Rendezvous.Entry_List;
      Select_Mode               :
RTS_Rendezvous.Modes;
      Index                     :
RTS_Rendezvous.Entry_Index
      Entry_Params              :
System.Address;
      Saved_Priority            : Integer;
      Resource_To_Get           :
Resources.Resource;
-- Address of resource object on which the
-- task is in a timed wait.

      Suspend_Semaphore         :
System.Access_Descriptor;
-- AD to Semaphore Object used in task
-- suspension (synchronous).

      Holding_Semaphore         :
System.Access_Descriptor;
-- AD to Semaphore Object used in task
-- suspension (asynchronous).

      Holding_Disabled_Count : Integer;
-- Count of nested calls to enable/disable
-- holding.

      Saved_Process_Controls :
Process_Controls_Blocks.Process_Control_Word;
   end record;
```

4.1.2. Task Creation and Activation

Tasks come into being in two stages, creation and activation. Tasks are created by the elaboration of declarations of objects that are either defined as tasks or contain an instance of a task type. Both tasks and task type instances are treated identically by the exec. Task creation involves the following steps:

Storage manager is called to allocate a record for the task control block (TCB). Most of the fields of the TCB, including the fields of the **process object** are initialized with information provided in the compiler generated call to create task. Specifically the following process object fields are set:

- **Dispatch port AD** is set to the same dispatch port which is bound to the Processor Control Block.
- **Primary Environment Table AD** if the task needs to make subsystem calls. The compiler provides this as part of the tasks init state information.
- **Region 0 AD** and **Region 1 AD** with read and write rights are set to indicate where the tasks code and data reside. This is also part of the tasks init state.
- The **process controls word** for execution mode, time slicing (default is off), task priority, process state, and preemption.

Port objects are allocated for each entry of the task. The Entry Criteria selected (either for this task, or globally for all tasks) determine if a FIFO Port or a Priority Port is used for the entry queue. An array of AD's is created for these port objects

and the address of this is set in the `Entry_Queues` field of the TCB. A Port object is also allocated as the `Ref_Dependent_Port` to maintain a list of completed tasks that are reference dependent on this task.

Semaphore objects are allocated, ADs are created and stored in the following fields of the TCB: `Act_Semaphore, Delay_Semaphore, Select_Semaphore, Suspend_Semaphore,` and `Holding_Semaphore`. The usage of these are discussed in the following sections.

Stack space for the task is allocated as an object. An AD to this is created and set as the **Region 2 AD** in the process object. An initial frame is set up in the stack space, with a dummy previous frame pointer (R0), Stack Pointer (R1) to the start address of the stack (64 bytes from the start of the frame), and Return Instruction Pointer (R2) to the entry point address of the task.

The following Ada tasking related initializations are performed:

- Task `state` is set to *initial*.
- `Activator` is set to the task id of the task creating this TCB and this TCB is linked to the activation chain of the creating TCB.
- `Master` and `Parent_task` fields are initialized. This will be null for library tasks.
- Priority of the task (in the process controls field of the process object) is set to the specified priority or a predetermined default value.

Finally, the created TCB is made an object (i.e. incorporated into the object table), and AD is created to this object and returned to the caller as a task id.

Task activation is done at the **begin** that follows a declarative part that created the task. An activation chain is local to a declarative part. The compiler generated code declares an activation chain at the begin of the declaration part and initializes it to null. Every time a task is created, the exec links the newly created TCB to the given activation chain. The TCB maintains a pointer to the activation chain and a link to the next TCB in the same activation chain. At the end of the declarative part (just before the first instruction of the block is to be executed) the compiler calls the procedure **Activate_Tasks** with the activation chain as parameter. The activating task first raises its priority to the highest priority, so that it may not be preempted while scheduling the tasks that it is activating. It then performs the following for each of the task on the chain:

- If the task has not completed elaboration, then the exception **Program Error** is raised.
- The ***schedprcs*** instruction is executed with the task id (AD). This sends the process/task to the dispatch port, where it is scheduled for execution at the front of its priority queue. Note that even though preemption is enabled, the high active priority of the activating task nullifies that effect.

After all tasks have been scheduled, the activating task changes it priority to its original (base) priority and executes the ***wait*** instruction on the `Act_semaphore` in a loop for `Num_Tobe_Activated` times.

The procedure **Complete_Activation** is generated by the compiler at the end of the declarative part of the body of a task. This procedure marks the tasks status as *activated* and executes the ***signal*** instruction on the activator tasks `Act_Semaphore`. The activator task will exit the wait loop after the last activated tasks signal, mark its status as *completed-activation* and then continue execution.

4.1.3. Task Termination

Tasks come to an end in two stages. First they complete and then they terminate. Once a task is completed it does not run again. A completed task becomes terminated when all tasks that depend on it either complete or agree to terminate.

The procedure **Complete_Task** is called as the last step of a task, whether the task completes normally or through an exception. There is no return. If the task has dependents, the exec performs all actions required for Complete_Master. In addition the exec performs all actions required on completion of a task. Some of these actions are:

- Check for pending calls on the task and

raise **Tasking Error** in any such calling task. This is done by checking the queue state (Port Lock and Status) field in each of the Port Objects (entry queues). The status of tasks blocked at the port are marked *rendezvous-failed* and they are unblocked by a ***send*** instruction.

· Destroy all tasks that are reference dependent on any block of this task. Those tasks were already completed, and are waiting as messages on the `Ref-_Dependents_Port` of this task. The ***condrec*** instruction is executed on this port. Any message received corresponds to the task id of a previously completed task. The following is done for each message received:

  · Deallocate the Region 2 object associated with the task.

  · Deallocate the environment table object (if any) associated with the task.

  · Deallocate the task control block.

  · Invalidate the object descriptor for this TCB. (Usage of dangling Task ids to these TCB will fault the system).

· Mark the tasks state as *terminated*. If this task is not a library task, remove it from the master dependency chain.

· Terminate any tasks that have become ready to be terminated through completion of this master.

· If the activator of this task has not completed activation, set its status as *failed* and ***signal*** its `Act_Semaphore`.

· The stack space (Region 2) is not deallocated at this point. The task includes itself to the reference chain of the parent task. (The reference dependent TCB's are maintained even after completion to provide the capability to test their 'TERMINATED and 'CALLABLE attributes.) The task ends it execution by executing the ***sendserv*** instruction on the `Ref_Dependents_Port` of the parent task.

4.1.4. Task Entry Calls

The compiler translates unconditional and un-timed entry calls to calls to runtime procedures: Call_Simple and Call_Conditional. A Rendezvous is possible if the called task is suspended at an **accept** or at a **select** statement and is waiting at the entry being called. The input parameters to the entry call procedures are the task id of the task being called, the entry index of the entry being called and the virtual address of any parameters to the entry. The following is done to try to initiate a rendezvous:

· The called tasks state and status is checked to make sure that it is callable, i.e. it has not completed, terminated or in any way abnormal. If the task is not callable, the exception **Tasking Error** is raised.

· The entry index and parameter address are saved on the called tasks TCB (`Index`, and `Entry_Params` field).

· If the state of the called task is *selecting* (unconditional, timed, or terminable), and the entry being called is on the select list as an open accept statement, then the following is steps are taken:

  · If the select mode is *timed*, the time manager is called to remove the called task from the time queue.

  · The called tasks priority is saved in the calling tasks TCB. If the called tasks priority is lower than that of the calling task, then its priority is changed (in the **process controls** filed of its process object) to the higher priority.

  · The partners list is updated to include the calling task.

  · The called tasks state is made *ready* and the calling (current) tasks state is set to *engaged*.

  · The called (selecting) task is unblocked by executing the ***signal*** instruction on it `Select_Semaphore`.

· If the called task was not selecting then the state of the calling task is marked as *calling-entry-unconditionally*.

· In either of the above two cases, the following is also done:

  · The AD to the entry queue (port object) corresponding to the entry being called is fetched from the called tasks entry queues list.

- The calling task executes the ***sendserv*** instruction with the entry queue port AD. This suspends the calling task and sends it as a message to the entry queue port. If no process is blocked at the port (i.e. the calling task has not executed an accept or select call) then sendserv enqueues the calling process as a message at the end of the appropriate message queue. If any process are blocked at the port (i.e. the called task is blocked on an accept statement) then sendserv unblocks that process and stores the calling process AD in that process objects received message.

For a conditional entry call, the calling task first checks the entry queue to see if the task being called is blocked on an accept for this entry. It fetches the AD to the port object corresponding to the entry being called. It locks the port object and examines the queue status. If a process is blocked at the port, it unlocks the port object and executes the ***sendserv*** instruction. If not it returns to the caller as an unsuccessful rendezvous.

Timed entry calls are translated to the runtime procedure **Call_Timed**. This procedure checks the status of the called entry queue port just as in a conditional entry call. If the rendezvous is not possible, the time manager is called to include this task in the delay list for the specified duration in the timed entry call. The state of the task is set as *timed-entry-call*. The ***sendserv*** instruction is now executed. The calling task is thus suspended. This task resumes execution in either of two ways. The delay may expire before the called task synchronizes for the rendezvous. In this case the timer interrupt handler, realizing that the task whose delay expired is in state *timed-entry-call*, traverses the message list pointed to by the `Called_Entry_Port` field of the TCB and removes this task from the list. It marks the tasks status as *delay-expired*. The interrupt handler then executes a ***schedprcs*** instruction with the message (task id) that was delinked from the port.

The second way is when the rendezvous synchronization takes place before the delay expires. The task that accepts the entry, realizing the calling task is in state *timed-entry-call*, calls the time manager to remove that task from the delay list. This is done prior to executing the rendezvous code. Upon rendezvous completion, the calling tasks status is marked as *rendezvous-completed* and is unblocked by a ***schedprcs*** instruction.

The code that follows the ***sendserv*** instruction in the Call_Timed procedure checks the tasks status to see if it was unblocked because of the delay expiration, or due to the completion of the rendezvous. It then returns a boolean value for Rendezvous_Successful.

4.1.5. Accept and Selective Waits

The compiler generates a call to **Accept_Call** when it sees **accept** statements outside of select statements. (Accept statements with no code for the entry block are optimized through the Trivial_Accept call. This is discussed at the end of section 4.2.1) The input parameter to the Accept_Call procedure is the number of the entry being accepted. The rendezvous synchronization is attempted as follows:

- The AD to the entry queue port corresponding to the entry being accepted is fetched from the accepting tasks TCB. A ***receive*** instruction is executed with this port AD. The receive instruction receives a message (task id of a task blocked on an entry call) from the port requested. If the port contains no messages (i.e. no callers for this entry), then the executing process (accepting task) blocks until a message is received.
- When this task gets to execute, the rendezvous has been initiated. The received message is the task id of the task calling this entry. The state of the calling task now changes. If it was in *timed-entry-call* state, it is removed from the time queue. Its state is now set to *engaged-in-rendezvous*.
- The priority of the accepting task is modified to be the higher of the calling task and itself. This is accomplished executing the ***modpc*** instruction. The old priority of the called task is saved in the `Saved_Priority` field of the calling task.
- The rendezvous partners list of the called task is updated to include the calling task.

· The entry parameter block address is copied from the calling tasks TCB to the accepting tasks TCB. The compiler needs to generate code to transfer the entry parameters in a form that is consistent with normal procedure calling conventions before giving control to the generated code for the entry block.

Select statement code begins with evaluation of guards for the accept alternatives. Compiled code builds a list of open alternatives in a data structure called an entry list. The valid elements in this list are only for those entries whose guards evaluate to 'open'. The **Selective_Wait** procedure is called to process the select list. The 'mode' parameter to this procedure indicates the kind of select statement. This procedure first saves the select list and mode in the TCB (`Select_List, Select_Mode` fields). It then examines its entry queue to see if any task is waiting for any open entry. If so, immediate rendezvous is possible. If more than one is available, one is selected based on the `Select_Criteria` field specified in the tasks `Task_Type_Descriptor`. The rendezvous synchronization mechanism is similar to the single accept call explained above. The index to the accepted entry and the rendezvous parameter address is returned to the compiled code, which then builds the parameter block and branches to the appropriate entry block code.

If there are no pending entry calls on any of the open alternatives, then the following is done:

· If the select mode is **simple**, the task marks its state as *selecting-unconditional* and suspends itself by executing the ***wait*** instruction on the `Select_Semaphore`. When it resumes execution (due to an entry call signalling this semaphore) it gets the entry index from its TCB and executes the ***receive*** instruction on the corresponding entry queue port. It then returns the selected entry and parameter to the compiled code.

· If the select mode is **delay** (and the delay duration is positive), the state of the task is marked *timed-select*. The time manager is called to place this task on the delay list. The task then suspends itself by executing the *wait* instruction on the Select_Semaphore. This task resumes execution in either of two ways. The delay may expire before the selecting task synchronizes for the rendezvous. In this case the timer interrupt handler, realizing that the task whose delay expired is in state *timed-select*, fetches the `Select_Semaphore` AD from its TCB and executes the ***deqsema*** instruction to remove the process from the semaphore. It marks the tasks status as *delay_expired*. The interrupt handler then executes a ***schedprcs*** instruction with the message (task id) that was dequeued from the semaphore.

The second way is when the rendezvous synchronization takes place before the delay expires due to an entry call on one of the open select entries. The task that calls the entry, realizing that the called task is in state *timed select*, calls the time manager to remove that task from the delay list. It then unblocks the selecting task by executing the ***signal*** instruction on its `Select_Semaphore`.

The code that follows the ***wait*** instruction in the Selective_Wait procedure checks the tasks status to see if it was unblocked because of the delay expiration, or due to the completion of the rendezvous. It then returns either a null index (if the delay expired) or the index of the called entry and its entry parameter address.

· If the select mode is **terminate**, then the state of the task is marked *terminable-select*. An attempt is made to release the master block on which this task depends on. If that is not successful, then this task cannot complete now. It suspends itself by executing the ***wait*** instruction on its `Select_Semaphore`. This process can only be resumed (if at all) by an entry call on any of its open alternative. It then returns the called entry index and parameter address.

4.1.6. Rendezvous Completion

At the end of the code of an accept entry block, the compiler generates a call to **Complete_Rendezvous** to reactivate the task that

called this entry. This procedure accomplishes the following:

- The task id of the rendezvous partner (calling task) is fetched from the top of the `Partners` list pointed to from the TCB. It is also delinked from this list.
- If the called tasks saved priority is different from its current priority, then it is changed to its original priority by executing the ***modpc*** instruction.
- The calling tasks state is made *ready* and is unblocked by executing a ***schedprcs*** instruction.

4.1.7. Task Abortion

One or more tasks may be aborted by the ***abort*** statement. The compiler generates a call to the **Abort_Tasks** procedure with the list of task ids. Each task in the list is aborted in the following way:

- If the task being aborted is not already terminated or being aborted, then the callers list and partners list is traversed and the exception **Tasking Error** is propagated in each of those tasks by diverting the control to the (compiler provided) interface procedure **Raise_Exception**. This is accomplished by first allocating and building the `Pre_Call_State` from the saved global registers in the process object. Then the saved global registers and the current frame are modified to reflect parameters and entry point address of the procedure Raise_Exception. The state of the task is made *ready*. The tasks are unblocked in a manner consistent with their state, and would thus enter the compiler's exception handling mechanism.

  If the task being aborted has not completed activation, all unactivated tasks are terminated. The abort_task procedure is recursively attempted for all child tasks of this task being aborted.
- If the task being aborted is in the process of activating or engaged in a rendezvous, mark its status as *aborting*.
- If the task being aborted is blocked on an entry call, it is logically delinked from the appropriate port.
- If the task being aborted has not completed activation, its activator's status is marked as *failed-activation* and the ***signal*** instruction is executed on the activator tasks `Act_Semaphore`.
- Finally, if there are no child tasks to this task, it is terminated.

4.1.8. Task Attributes

Task attributes refers to the user modifiable attributes of tasks that the exec supports, namely: priorities, and time-slice scheduling of tasks. (The Ada predefined task attributes, 'Count, 'Callable, and 'Terminated are directly deciphered from the task entry queues, and state information.)

The priority of a task may be changed by either executing the ***modpc*** instruction (if it is for the current executing task) or by modifying the value in the process controls field of the process object of the specified task.

With time-slice scheduling, the processor works on each process for a set duration, called a time-slice. Six fields in the process object support time-slice scheduling: the residual-time-slice, next-time-slice, and execution-time fields; and the timing, time-slice, and time-slice-reschedule flags in the process controls. Each task can have a different time-slice value, ranging from 16 ticks to 2^{24}-1 ticks. The exec supports the CIFO interface **Set_Time_Slice** procedure. The time slice for the task specified is set in the next-time-slice field of the process object. The **Turn_Off_Time_Slice** procedure disables time slicing by turning off the 'timing' bit in the process controls field of the process object of the specified task.

4.1.9. Task Suspension/Holding

While there is no support in the Ada language for task suspension, this feature is provided through the CIFO task suspension mechanism (packages Task_Suspension, Two_Stage_Task_Suspension, and Asynchronous_Task_Holding). The suspend, resume mechanism is implemented by using a semaphore object associated with each task (the `Suspend_Semaphore` field in the TCB for the synchronous case and the `Holding_Semaphore`

for the asynchronous case). The task waits at its associated semaphore when it is suspended. The two cases of suspension are discussed below:

- In the synchronous case, the task being suspended is the current executing process. The status of the task is checked to see if it is 'suspendable' (true by default in the general case, and true if the procedure **Will_Suspend** is in effect for two stage task suspension). If not suspendable, then this is just a latching operation and the task is not suspended. Otherwise, the `Suspend_Semaphore` AD is fetched from the TCB. The tasks state is marked as *Suspended* and the ***wait*** instruction is executed, specifying the semaphore AD. This saves the task and blocks it at the semaphore. The processor dispatches the next ready task.
- In the asynchronous task holding mechanism the task being suspended may not be the executing task. In this case the `Holding_Disabled_Count` is fetched from the TCB. Only if it is zero will the task be suspended. The suspension is achieved using the architectures **event fault request** mechanism. This is a software interrupt mechanism via an Event Notice Fault. The event-request-flags in the process-notice-and-lock field of the process object of the task being suspended are set. The ***stvos*** (store virtual ordinal short) instruction may be used to set the flags. When this task gets to execute, the processor automatically raises the event notice fault. The fault handler then loads the `Holding_Semaphore` AD from the TCB and executes the ***wait*** instruction. When the process is resumed it returns from the fault handler and continues execution of the task.

Task resumption mechanism is the same for suspended and asynchronously held tasks. The state of the task being resumed is checked to ensure that the task is suspended. The appropriate semaphore AD is fetched from the TCB of the task being resumed (`Suspend_Semaphore` for a Resume_Task operation and the `Holding_Semaphore` for the Release_Task operation). The ***signal*** instruction is executed, specifying the semaphore AD. This will unblock the task and schedule it for execution.

Support is also provided for keeping the a current executing process bound to the processor. This is needed for critical regions of code in which the processor cannot be involuntarily reassigned to another task. The **Disable_Dispatching** (CIFO procedure) is used at the beginning of the critical section of code. It does the following:

- The current process controls is saved in the TCBs `Saved_Process_Controls` field.
- The status of the task is marked as *Dispatching_Disabled.*
- The ***modpc*** instruction is issued to disable timing and raise the process priority to the highest task priority in the system.

The procedure **Enable_Dispatching** is used at the end of the critical section to undo the effect of Disable_Dispatching as follows:

- The saved process controls word is fetched from the TCB. The ***modpc*** instruction is issued with this value to restore the process to its original state.

4.2. Interrupt Management

During initialization the exec configures the interrupt mechanism to function in the "dedicated mode" i.e. each external interrupt pin is a separate interrupt source. The processor control block points to the interrupt table, and interrupt stack, both of which are located in Region3. Initially interrupts are masked and any pending interrupts are cleared in the Special Function Register SFR0. The interrupt vectors are initialized in the Interrupt Vector Registers 0 and 1. The interrupts are enabled by setting all bits in the Interrupt Enable Special Function Register SFR1.

The exec supports interfacing interrupts to Ada tasks in two ways. These are specified in the MRTSI packages RTS_Interrupts and RTS_Queued_Interrupts. The former provides an efficient mechanism for an interrupt handler. The latter is a more generalized form of Ada task with interrupt entries. The exec also supports masking, unmasking, and fetching the mask of specific interrupts as specified in the CIFO package Interrupt_Management.

4.2.1. Restricted Interrupt Handler Task

This mechanism supports the direct execution of task interrupt entries. However, the following restrictions are imposed on such a task:

- The task has only one interrupt entry.
- All of the code in the task is inside a single accept statement, which could be in a simple (infinite) loop.

The compiler recognizes such an interrupt task. The code for the accept body is generated as a procedure that the interrupt handler could call directly. A task is not created for this interrupt task. Regular runtime calls are not generated from within the accept block.

The procedure **Bind_Handler** causes the exec to arrange for any subsequent occurrence of the specified interrupt to transfer control to specified address of the handler. The previous handler (if any) is saved. If the input linear address is not a Region 3 address, then the **Interface Error** exception is raised. The compiler recognizes an interrupt handler task and compiles the accept block code into Region 3. The compiler also saves all global and floating point registers used in the handler or other subprograms that it calls. This is required because the hardware interrupt mechanism only saves all local register sets, and gives control to the handler under the context of the current executing process. Note that if the handler has too many nested calls, the cached local register sets may all get used up resulting in the processor flushing all local register sets to memory. This would cause significant performance loss in interrupt handling.

The procedure **Unbind_Handler** undoes the effect of the last call to Bind_Handler for the interrupt at hand. In particular it reinstates whatever binding to handler existed before that call. (Only one old handler value is saved; they are not stacked.) This procedure is called at the completion of the interrupt entry task. If the accept block was not in a loop, then the compiler generates this call at the end of the code for the accept block, to dissociate the interrupt entry from the interrupt. This procedure is also called if the interrupt task completes because of the occurrence of an exception in the accept block.

The exec also supports the call of the interrupt entry from a regular task. The compiler, realizing that the entry being called is an interrupt entry, generates a call to **Software_Interrupt**. This procedure causes the executing task to be interrupted, and control transferred to the handler (if any) for Interrupt. The calling task is eventually resumed at the point following the call. The exec uses the architectures **Inter Agent Communication** (IAC) mechanism to generate a software interrupt for the specified interrupt vector. The interrupt handler is given control in the same way as a hardware interrupt.

The only rendezvous allowed from an interrupt entry block is a simple entry call to an entry associated to a simple accept statement with no sequence of statements or parameters. The compiler translates the entry call to the special procedure **Trivial_Call**. This is implemented as a conditional entry call, and all interrupts are disabled. In no case must the calling task wait. This is accomplished by executing the *condrec* instruction to the port associated with the called entry. The condition code indicates if the synchronization took place or not.

The corresponding simple *accept* statement is translated to a call to the procedure **Trivial_Accept**. The task that contains the accept statement must be a full fledged Ada task (not a restricted handler). Upon calling this procedure, the task waits until it is signaled to go on via Trivial_Call. The accepting tasks state is marked as *accepting-entry* and suspends itself on the port associated with the entry using the *sendserv* instruction.

4.2.2. Unrestricted Interrupt Handler Task

In the unrestricted case, the interrupt mechanism merely notifies the accepting task of the event. Normal Ada task priorities govern the execution of entries. The interrupt task is a full fledged Ada task, some or all of whose entries are interrupt entries. Since the mechanism of entry calls is message passing using ports, it is not possible for the accept to distinguish between an entry call from an interrupt handler and Ada entry call. This imposes the restriction that **interrupt entries may not be called from regular tasks.** The exec implements the interrupt entry calls as conditional entry calls.

The procedure **Associate_Interrupt** notifies the exec that the specified interrupt is to be associated with the given entry of the given task. Every occurrence of this interrupt is processed as a **conditional entry call** to the given entry. Thus, when the interrupt occurs, if the called task is waiting on a matching *accept* statement, the rendezvous takes place. As in the case of the restricted interrupt handling mechanism, the previous handler is saved by the exec. The task identifier and entry index are also saved in a global data structure that has a one-to-one mapping to the interrupt table. The interrupt vector is set to call an internal procedure which in turn calls the standard interface procedure **Call_Conditional** to initiate the rendezvous. The task control block of the interrupt task is also modified to indicate that the entry type is an interrupt entry. Thus processing at the end of a rendezvous for the interrupt entry omits all operations related to the calling task. Passing interrupt data as parameters to the interrupt entry may be achieved by a user provided function in the form *User_Interrupt_N_Routine*. This is tailored to the specific interrupt, and may build a parameter block for the accept entry from the interrupt data. If such a procedure exists, the exec calls it prior to initiating the rendezvous. The address returned from this user function is passed as the parameter address to the entry call.

4.3. Time Management

The Ada supported time management functions include: delaying a task for a particular time period, and fetching the system time. A periodic external timer interrupt is used to implement delays and the system time. A task may be delayed for one of the following reasons: execution of the **delay** statement; timed entry call on an entry that is not accepting; a **select** statement with a **delay** in the **else** part.

The **delay** statement is implemented by calling the time manager function to add the task to the delay list. The tasks state is marked *delayed*. The task then suspends itself by executing the ***wait*** instruction on the `Delay_Semaphore` associated with its TCB. When the delay expires, the timer interrupt handler, realizing that the task was delayed, wakes up the task by executing the ***signal*** using the `Delay_Semaphore` AD.

The other two forms of delaying a task are discussed earlier in sections: 4.1.4 and 4.1.5.

The timer interrupt handler performs two duties:

- Increment the system's date/time values.
- Service a list of delays, corresponding to tasks waiting for a timed delay to expire.

The system date/time is stored as a long ordinal number of timer ticks since a specific base date/time. The timer interrupt handler simply increments the value when it executes.

A list of delays is maintained as a linked list of records in chronological order, with next delay to expire at the head of the list. Each record contains the following fields:

- Task id. The AD of the task that is delayed. When the delay expires, the timer interrupt handler wakes up this task.
- Delta. The number of timer ticks between the previous tasks expiration time and this tasks expiration time. For the first record, this field contains the number of timer ticks between the current system time and this tasks expiration time.
- Link to next record. Null value indicates the end of the list.

The timer interrupt handler services the delay list in the following way. If the delay list is not empty, the first records delta is decrimented. If the new delta is zero, then the tasks delay has expired. This record is now delinked from the delay list. The TCB of the associated task is accessed. Depending on the state of the task, (*delayed, timed-entry-call, timed-select*) the appropriate action is taken as explained in previous sections. The same action is repeated for each subsequent record with zero delta.

The time manager supports functions to add a task to the delay list, and to delink any task from the list.

4.4. Storage Management

The storage management functions are defined in the (MRTSI) package RTS_Storage_Management. By default, all dynamically created Ada objects (except tasks) are allocated on the linear address space, and the access

values are linear addresses. However, objects may be allocated in the virtual address space by the use of special pragmas (viz. a *pragma access_descriptor* associated to an Ada access type). Thus the storage manager supports memory management in both the linear and virtual address spaces. Linear memory management allocates blocks out of Region 3. Collection_Id and block addresses are linear addresses. Virtual memory management allocates memory out of the application heap area. The collection_ID in this case is an AD and block addresses are virtual addresses.

The function New_Collection reserves a storage collection as part of the elaboration of an access type definition. An internal data structure is maintained which includes the extent of this collection and the block size. If the allocation descriptor is *linear* the collection resides in Region 3 and a linear address represents the collection_Id. If it is *virtual*, the collection resides in the virtual address space. An AD is created for the collection and is returned. For protection, the linear addresses corresponding to the extent of the virtual collection is marked as invalid. A **Storage Error** exception is raised if the memory manager cannot reserve the specified amount of storage.

Allocation and deallocation of blocks within collection may be implemented using standard algorithms like first-fit, best-fit, etc.

4.5. Exception Propagation

Exception handling is mainly the responsibility of the compiler. However the Ada executive needs to be able to raise exceptions in Ada tasks. The hardware fault handling mechanism of the exec also needs to map the faults that it cannot internally handle to Ada exceptions. This requires an interface between the compiler's exception handling mechanism and the Ada Executive. The exception handler also may propagate an exception to a task. Thus the exec needs to be able to identify exceptions. The MRTSI package Compiler_Exceptions defines the standard Ada exceptions and the interface procedures **Raise_Exception**, and **Notify_Exception**. The compiler should provide the implementation of these procedures.

The Ada executive initializes the first sixteen entries of the fault table with address of runtime procedures. These procedures are located in Region 3 and all entries in the fault table are local procedure fault table entries. For those fault types/subtypes that map to compiler exceptions, the exec builds the parameters and makes an inter-domain call to the compiler interface procedure *Notify_Exception*. The exec switches to supervisor mode before it processes the other fault conditions. The fault conditions that map to the predefined Ada exceptions are listed below:

Constraint Error: Arithmetic Fault, Constraint (range) Fault, Floating Point Fault.

Program Error: Attribute Fault, Constraint (invalid AD) Fault, Descriptor Fault, Operation Fault, Protection Fault, Type Fault, Virtual Memory (invalid object descriptor) Fault.

Storage Error: Control Stack (overflow) Fault.

4.6. Time Critical Section

The Ada executive provides the ability to ensure that a given segment of code is executed without preemption and with minimal interruption. This is a CIFO extension to the standard Ada runtime. The implementation of this feature is exactly the same as the Disable_Dispatching and Enable_Dispatching interfaces explained in section 4.1.9.

4.7. Resources

The resource is an implementation of a synchronization object that real-time applications can make use of to efficiently control access to a shared resource. The architecture defined semaphore object is used directly to model a resource. The following declaration defines a resource object.

```
type Resource_Block is
   record
     Semaphore_AD : system.Access_Descriptor;
     Capacity     : Capacity_Range :=
Capacity_Range'LAST;
     Count        : Counter_Range := 0;
     Name         : String (1..Name_Range);
   end record;
type Resource is access Resource_Block;
```

Resource creation involves allocating a record for the Resource_Block, allocating a **semaphore object**, creating an AD, and storing it

in the Resource_Block. The access to the Resource_Block is returned to the caller of the create operation.

A task may unconditionally try to secure a resource. This is implemented by first decrementing the count associated with the resource block using the ***atadd*** instruction. If the new count is less than zero (i.e. the resource was already busy) then the current task is blocked by executing a ***wait*** instruction on the semaphore associated with the resource. Its state is marked as *unconditional-get-resource.*

A timed get of a resource is implemented by adding the task to the delay list, if the resource is not available. The tasks state is marked as *timed-get-resource*. The address of the resource block is saved in the TCBs `Resource_To_Get` field, and then the task suspends itself by executing a ***wait*** instruction. The code following this checks the task status to see if it was resumed by a ***signal*** on the semaphore, or by the delay expiration, and returns the appropriate boolean value to the caller to indicate if the resource was secured or not.

The release operation on a resource is achieved by incrementing the count field of the resource using the ***atadd*** instruction. If the new count is less than or equal to zero (ie. there are tasks waiting on the semaphore) then a ***signal*** instruction is issued on the semaphore associated with the resource.

5. Conclusion

This paper provides a comprehensive design strategy for building the executive component of an Ada Runtime System for the 80960 extended architecture. Adopting MRTSI and CIFO interfaces to specify the functionality provides a standard interface to both the compiler and the Ada application. While this is an optimal design that closely matches the hardware capability, there are avenues for optimizing the implementation. One instance is the number of semaphores associated with a task. Careful analysis of interaction of features, may lead to a reduction in the number of semaphores that a task needs.

The standard interface facilitates compiler and runtime system development and the kernel implementation provides the best exploitation of architecture features. Programs developed using different Ada Compilers for the 80960 that interface to the Ada Executive can be easily integrated.

References

1. Ada Runtime Environment Working Group MRTSI Task Force, "Model Ada Runtime System Interface," in Ada Letters Volume X Number 5, May/June 1990.

2. Ada Runtime Environment Working Group Interfaces Subgroup, "Catalogue of Interface Features and Options for the Ada Runtime Environment, Release 3.0" Association for Computing Machinery Special Interest Group for Ada, July 1991.

3. Intel Corporation. i960™ Extended Architecture Programmer's Reference Manual.

4. Intel Corporation. Military i960™ MX Microprocessor Technical Overview (order number 271194-001).